CHAPTER 1

SSA Reconstruction

S. Hack

Progress: 60%

Structural reordering in progress

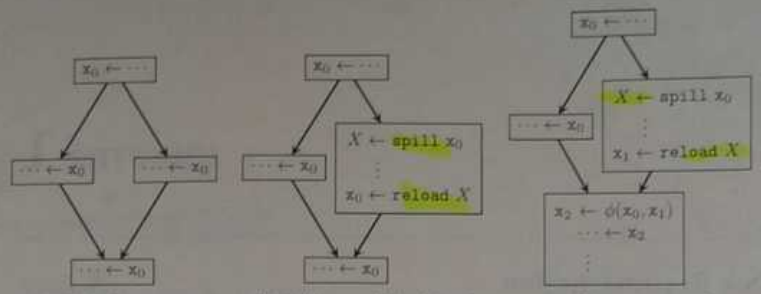
1.1 Introduction

Some optimizations break the single-assignment property of the SSA form by inserting additional definitions for a single SSA value. A common example is live-range splitting by inserting copy operations or inserting spill and reload code during register allocation. Other optimizations, such as loop unrolling or jump threading might duplicate code, thus adding additional variable definitions, *and* modify the control flow of the program. Let us first go through two examples before we present algorithms to properly repair SSA.

1.1.1 *Live-Range Splitting*

In Figure 1.1b, our spilling pass decided to spill a part of the live range of the variable x_0 in the right block. Therefore, it inserted a store and a load instruction. This is indicated by assigning to the memory location **X**. The load however is a second definition of x_0, hence SSA is violated and has to be reconstructed as shown in Figure 1.1c. Furthermore, This figure shows that maintaining SSA also involves placing ϕ-functions.

Such program modifications are done by many optimizations. Not surprisingly, maintaining SSA is often one of the more complicated and error-prone parts in such



(a) Original program (b) Spilling x_0, SSA violated (c) SSA reconstructed

Fig. 1.1 Adding a second definition

optimizations; owing to the insertion of additional ϕ-functions and the correct redirection of the variable's uses.

X but : relayer tâche à l'IR

1.1.2 *Jump Threading*

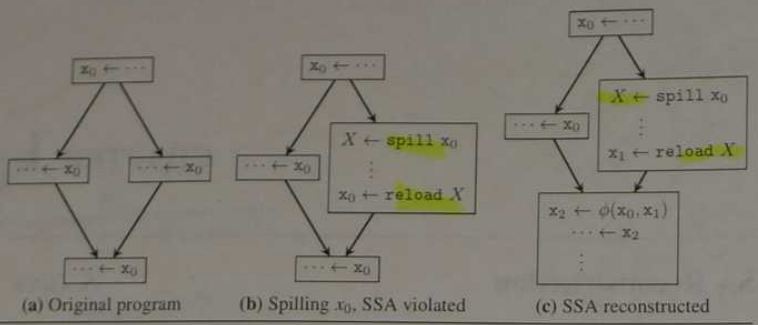
Jump threading consists on removing control flow operations by duplicating some code.

~~Jump threading is a transformation performed by many popular compilers such as GCC and LLVM~~. Jump threading is applied to enable more expressive optimizations. Consider following situation: A block contains a conditional branch that depends on some variable x. In the example shown in Figure 1.2, the conditional branch tests, if $x = 0$. Assume that the block containing that conditional branch has multiple predecessors and x can be proven constant for one of the predecessors. In the example below, this is shown by the assignment $x_1 \leftarrow 0$. Jump threading now partially evaluates the conditional branch by directly making the corresponding successor of the branch block a successor of the *predecessor* of the branch. To this end, the code of the branch block is duplicated and attached to the predecessor. This also duplicates potential definitions of variables in the branch block. Hence, SSA is destroyed for those variables and has to be reconstructed. In contrast to the examples above, the control flow has changed however which poses an additional challenge for an efficient SSA reconstruction algorithm.

x Souligner que ϕ qui est copié est dupliqué aussi
x Souligner que c'est du code correct non SSA, mais correct

1.2 General Considerations

In the following, we will discuss two algorithms. The first is an adoption of the classical dominance-frontier based algorithm. The second performs a search from the variables' uses to the definition and places ϕ-functions on demand at appropri-



(a) Original program (b) Spilling x_0, SSA violated (c) SSA reconstructed

Fig. 1.1 Adding a second definition

optimizations; owing to the insertion of additional ϕ-functions and the correct redirection of the variable's uses.

X but : relayer tâche à l'IR

1.1.2 *Jump Threading*

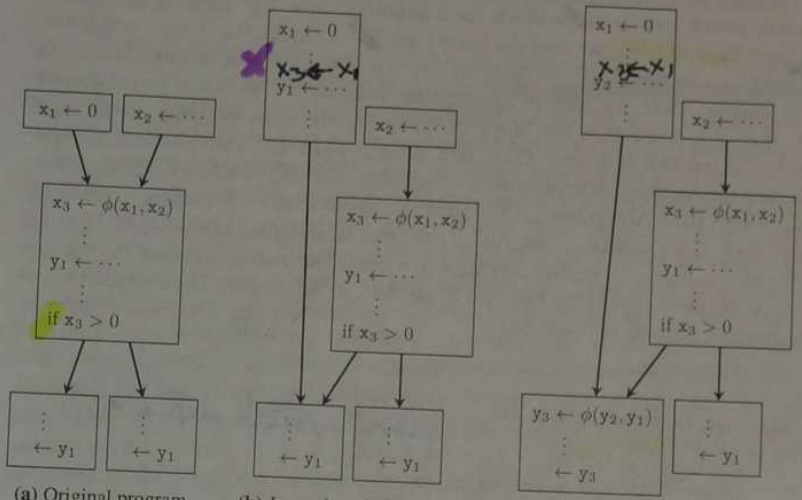
Jump threading consists on removing control flow operations by duplicating some code.

~~Jump threading is a transformation performed by many popular compilers such as GCC and LLVM~~. Jump threading is applied to enable more expressive optimizations. Consider following situation: A block contains a conditional branch that depends on some variable x. In the example shown in Figure 1.2, the conditional branch tests, if $x > 0$. Assume that the block containing that conditional branch has multiple predecessors and x can be proven constant for one of the predecessors. In the example below, this is shown by the assignment $x_1 \leftarrow 0$. Jump threading now partially evaluates the conditional branch by directly making the corresponding successor of the branch block a successor of the *predecessor* of the branch. To this end, the code of the branch block is duplicated and attached to the predecessor. This also duplicates potential definitions of variables in the branch block. Hence, SSA is destroyed for those variables and has to be reconstructed. In contrast to the examples above, the control flow has changed however which poses an additional challenge for an efficient SSA reconstruction algorithm.

x Souligner que ϕ qui est copie est dupliqué aussi
x Souligner que c'est du code correct non SSA correct, mais

1.2 General Considerations

In the following, we will discuss two algorithms. The first is an adoption of the classical dominance-frontier based algorithm. The second performs a search from the variables' uses to the definition and places ϕ-functions on demand at appropri-



(a) Original program (b) Jump threaded, SSA broken (c) SSA repaired

Fig. 1.2 Jump Threading

ate places. In contrast to the first, the second algorithm might not yield minimal SSA form (in Cytron's sense). However, it does not need to update its internal data structures when the CFG is modified.

We consider the following scenario: The program is represented as a control-flow graph (CFG) and is in SSA form and the dominance property holds. For the sake of simplicity, we assume that each instruction in the program only writes to a single variable. Due to the single-assignment property of the SSA form, we can then identify the program point of the instruction and the variable. An optimization/-transformation now violates SSA by inserting additional definitions for an existing SSA variable, like in the examples above. The original variable and the additional definitions can be seen as a single non-SSA variable that has multiple definitions and uses.

In the following, v will be such a non-SSA variable. In the pseudocode `v.defs` is the set of all definitions of v. A use of a variable is a pair consisting of a program point (a variable) and an integer denoting the index of the operand at the using instruction.

Both algorithms which we are going to present share the same driver routine (Algorithm 1). First, every basic block b is equipped with a list `b.defs` that contains all instructions in the block which define one of the variables in v. This list is sorted according to the schedule of the instructions in the block from back to front. Hence, the latest definition is the first in the list.

Then, all uses of the variable v are scanned. For every program point where v is used, search locally in its block for a definition of v. By scanning the block's list from back to front, we find the latest such use if one exists. If the variable has no definition in the block, we have to find the definition that reaches this block from *outside*. Here we have to differentiate whether the user is a ϕ-function or not. If it is a ϕ-function, the block of the use is the corresponding predecessor block. We use two functions `find_def_from_begin` and `find_def_from_end` that find the reaching definition at the beginning and end of a block, respectively. As can seen from Algorithm 1, `find_def_from_end` can be expressed in terms of `find_def_from_begin`. `find_def_from_end` additionally considers definitions in the block, while `find_def_from_begin` does. Our two approaches only differ in the implementation of the function `find_def_from_begin`. The differences are described in the next two sections.

```
proc ssa_reconstruct(var v):
  for d in v.defs:
    b = d.block
    insert d in b.defs according to schedule
    # latest definition is first in list

  for each use (u, index) of v:
    d = None

    # search for a local definition in the block
    for e in u.block.defs:
      if e.order < u.order:
        d = e
        break

    # no local definition was found, search in the predecessors
    if d == None:
      # if the user is a phi we have to start the search
      # at the end the corresponding predecessor block
      if u.is_phi():
        d = find_def_from_end(u.block.pred(index), ...)
      else:
        d = find_def_from_begin(b, ...)

    rewrite use at u to d

proc find_def_from_end(block b, ...):
  if not b.defs.empty():
    return b.defs[0]
  return find_def_from_begin(b, ...)
```

Algorithm 1: SSA Reconstruction Driver

1.3 Reconstruction based on the Dominance Frontier

This algorithm follows the same principles as the classical SSA construction algorithm ~~by Cytron et al. as~~ described in chapter (TODO: reference to the chapter). We compute the iterated dominance frontier (IDF) of v. This set is a sound overapproximation of the set where ϕ-functions must be placed (it might contain blocks where a ϕ-function would be dead). Then, we search for each use u the corresponding reaching definition. This search starts at the block of u. If that block is in the IDF of v a ϕ-function needs to be placed at its entrance. This definition will then serve as the reaching definition. The operands of the newly created ϕ-function will query their reaching definitions by recursive calls to `find_def_from_end`. The operands of that ϕ-function are then (recursively) searched in the predecessors of the block. If the block is not in the IDF, the search continues in the block's immediate dominator. ~~This is because in SSA, every use of a variable must be dominated by its definition~~[1]. ~~If the block is not in the IDF, the reaching definition is the same for all predecessors and hence for the immediate dominator of this block~~. Note that by rewiring the uses of several variables, some variables defined by ϕ-functions may not be used anymore. A dead code elimination pass after SSA reconstruction will remove these. Algorithm 2 shows this procedure in pseudo-code.

```
proc find_def_from_begin(block b, set of blocks F):
  if b in F:
    d = new_phi(b)
    i = 0
    for p in b.preds:
      o = find_def_from_end(p, F)
      set i-th operand of d to o
      i = i + 1
  else:
    d = find_def_from_end(b.idom, F)
  b.def = d
  return d
```

Algorithm 2: SSA Reconstruction based on Dominance Frontiers

1.4 Search-based Reconstruction

The second algorithm we present here is similar to the construction algorithm ~~that Click describes in his thesis [5]~~. Although his algorithm is designed to construct

[1] The definition of an operand of a ϕ-function has to dominate the according predecessor block.

SSA from the abstract syntax tree, it also works well on control flow graphs. Its major advantage over the algorithm presented in the last section is that it does neither require dominance information nor dominance frontiers. Thus it is well suited to be used in transformations that change the control flow graph. Its disadvantage is that potentially more blocks have to visited during the reconstruction. The principle idea is to start a search from every use to find the corresponding definition inserting ϕ-functions on the fly while caching the found definitions at the basic blocks. This is similar to the implementation of a data-flow analysis, that places the ϕ-functions. As in the last section, we only consider the reconstruction for a single variable called v in the following. If multiple variables have to be reconstructed, the algorithm can be applied to each variable separately.

We perform a backward depth-first search in the CFG to collect the reaching definitions of the variable in question at each block. We record the SSA variable that corresponds to the reaching definitions of v in the field `def` of that block. If the CFG is a DAG, all predecessors of a block can be visited before the block itself is processed (post-order traversal). Hence, all definitions that reach `b` can be computed before we decide whether to place a ϕ-function in `b` or not. If more than one definition reaches the block, we need to place a ϕ-function.

If the CFG has loops, there are blocks for which not all reaching definitions can be computed before we can decide whether a ϕ-function has to be placed. Recursively computing the reaching definitions for a block `b` can end up at `b` itself. To avoid infinite recursion, we create a ϕ-function `phi` without operands in the block before descending to the predecessors. Hence, if a variable has no definition in a loop, the ϕ-function placed in the header eventually reaches itself (and can later be eliminated). When we return to `b` we decide whether a ϕ-function has to be placed in `b` by looking at the reaching definition for every predecessor. If the set of reaching definitions is a subset of $\{a, x\}$ where x is the ϕ-function inserted at `b`, then ϕ-function is not necessary and we can propagate a further downwards. Otherwise, we place a ϕ-function.

```
proc find_def_from_begin(block b):
  if b.def != None:
    return b.def

  b.def = make_phi()
  reaching_defs = []
  for p in b.preds:
    reaching_defs += find_def_from_end(p)
  if phi_necessary(reaching_defs):
    set_arguments(b.def, reaching_defs)
  else:
    b.def = reaching_defs[0]
  return b.def
```

Algorithm 3: Search-based SSA Reconstruction

1.5 Conclusions

Some optimizations, such as loop unrolling or live-range splitting destroy the single-assignment property of the SSA form. In this chapter we presented two generic algorithms to reconstruct SSA. The algorithms are independent of the transformation that violated SSA and can be used as a black box: For every variable for which SSA was violated, a routine is called that restores SSA. The presented algorithms differ in the prerequisites and their runtime behavior:

1. The first is based on the iterated dominance frontiers like the classical SSA construction algorithm by Cytron et al. [2]. Hence, it is less suited for optimizations that also change the flow of control since that would require recomputing the iterated dominance frontiers. On the other hand, by using the iterated dominance frontiers, the algorithm can find the reaching definitions quickly by scanning the dominance tree upwards.
2. The second algorithm does not depend on additional analysis information such as iterated dominance frontiers or the dominance tree. Thus, it is well suited for transformations that change the CFG because no information needs to be recomputed. On the other hand, it might find the reaching definitions slower than the first one because they are searched by a depth-first search in the CFG.

Both approaches construct *pruned* SSA (TODO: cite other chapter), i.e. no ϕ-function is dead. One can also show that the first approach produces minimal SSA in the sense of Cytron et al. [2] whereas the second approach might create superfluous ϕ-functions in the case of irregular control flow.

This citation is to make chapters without citations build without error. Please ignore it: [?].

irréductible ? Pas seulement !
référence chapitre construct & copy-prop.
Ici ne simplifie que boucle interne

+ Références sur incremental computation (Dom Tree + DF)
+ référence thèse Click.